Preprint
UCRL-JC-140186-REV-1

MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs

J. M. May

This article was submitted to
15th Annual International Parallel and Distributed Processing Symposium, San Francisco, CA., April 23-27, 2001

January 8, 2001

U.S. Department of Energy
Lawrence Livermore National Laboratory

This work was performed under the auspices of the United States Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs

John M. May
Lawrence Livermore National Laboratory
johnmay@llnl.gov

Abstract

Hardware performance counters are CPU registers that count data loads and stores, cache misses, and other events. Counter data can help programmers understand software performance. Although CPUs typically have multiple counters, each can monitor only one type of event at a time, and some counters can monitor only certain events. Therefore, some CPUs cannot concurrently monitor interesting combinations of events. Software multiplexing partly overcomes this limitation by using time sharing to monitor multiple events on one counter. However, counter multiplexing is harder to implement for multithreaded programs than for single-threaded ones because of certain difficulties in managing the length of the time slices.

This paper describes a software library called MPX that overcomes these difficulties. MPX allows applications to gather hardware counter data concurrently for any combination of countable events. MPX data are typically within a few percent of counts recorded without multiplexing.

1 Introduction

For many years, CPUs have included registers that count various hardware events while code executes. Counted events typically include load and store requests, misses in Level 1 and Level 2 (L1 and L2) cache and in the translation lookaside buffer (TLB), floating-point instructions completed, and so on. The counters were intended to help the designers of the hardware and low-level software evaluate their systems, and their programming interfaces were not available to general users. However, since counter data can help application programmers tune their codes, hardware vendors have begun to publish the interfaces. Each vendor uses a different programming interface, and different CPU types (even from the same vendor) may count different types of events. At least two projects have developed uniform interfaces through which applications can access counters on different operating systems [1, 5, 6].

Many useful measures of application performance involve combinations of events. Cache utilization, for example, is the fraction of load requests that are satisfied from the cache. Unfortunately, some CPUs cannot count loads and cache hits (or misses) at the same time. Although most CPUs have multiple counters, different counters are designed to monitor different types of events. A single counter can monitor only one of type of event at a time, so two events that the hardware design has assigned to the same register cannot normally be counted simultaneously; these events are said to be "conflicting." In the PowerPC 604e architecture, for example, loads and L1 cache misses are conflicting events [4]. Even if all registers could count every type of event, a user might want to count more events concurrently than there are counter registers.

Time sharing (multiplexing) can solve these problems. One register counts different event types during separate time slices, and the multiplexing software can estimate how many times each type of event occurred if it can determine how long the counter spent monitoring each event type as a ratio of the total measurement period. One of the two main contributions of this paper is a technique for determining this ratio accurately under a wide range of measurement conditions, as described in Section 4.

Simple multiplexing algorithms produce adequate results when they measure one range of code at a time, but they don't work well for overlapping measurements. In multithreaded codes, for example, the user may wish to monitor the performance of several threads (running on the same or different CPUs) as they execute concurrently. Users may also wish to measure overlapping regions of code within a single thread. The second main contribution of this paper is a set of techniques for using counter multiplexing in multithreaded codes and other overlapped measurements. Section 3 briefly describes how applications initiate overlapped measurements, and Section 5 describes software that implements them.

These multiplexing and overlapping techniques are implemented in a library called MPX, which has been tested on an IBM RS6000/SP computer with SMP nodes that each

contain four PowerPC 604e CPUs. The nodes run IBM's AIX 4.3 operating system. However, the techniques are not specific to a particular processor or operating system. MPX is implemented on top of PAPI [5, 6], a system-independent interface for hardware performance counters. Section 6 compares the accuracy of MPX counts with nonmultiplexed counts gathered using PAPI.

2 Background and related work

Several hardware vendors publish interfaces to their systems' CPU performance counters. Examples include SGI's Perfex [7] and IBM's Performance Monitor API [3]. Compaq's ProfileMe [2] tool takes a different approach; it samples instructions as a code executes and determines what hardware events each of these instructions caused.

Some CPUs and operating systems can measure combinations of events concurrently without special software. Pentiums can count most events in any counter register, although multiplexing would still increase the number of events that can be counted concurrently. SGI's IRIX has built-in counter multiplexing: the system can switch the event that each register is counting every 10 milliseconds. It computes the total number of events of a given type by multiplying the observed count for an event type by number of event types that were multiplexed.

Each vendor interface reflects the unique capabilities of the hardware and operating system. Since many applications can run on multiple platforms, users of proprietary counter interfaces cannot develop portable instrumentation. Multiplatform interfaces, such as PCL and PAPI, can mask some of the differences between platforms.

The PCL [1] (Performance Counter Library) project is based at the Forschungszentrum Jülich (a German national research center). It defines a simple interface for performance counters on Alpha, MIPS, Pentium, PowerPC, and UltraSPARC processors. It allows nested measurements on overlapping regions of code, but it does not explicitly support multithreading or multiplexing.

The PAPI [5, 6] (Performance Application Programming Interface) project is based at the University of Tennessee, Knoxville. It supports the same processor families as PCL. PAPI allows measurements on overlapping regions, and it also works with threaded code, if there is thread support in the underlying hardware counter interface. PAPI was designed to accommodate multiplexing, but this feature was not in the initial implementation; indeed, the techniques described in this paper may become the basis for multiplexing in PAPI. At present, however, MPX is built on top of PAPI and uses only its standard interfaces.

Although multiplatform interfaces attempt to hide system dependences, they still work somewhat differently on each machine because of underlying hardware differences. Some event types may be countable on some systems but not on others, and corresponding events can have different meanings on different machines. For example, one system might count a floating-point multiply-add instruction as a single floating point operation while another counts it as two. Therefore, performance data gathered on different machines is not generally comparable.

3 Event sets

MPX both uses and imitates the PAPI programming interface. The two systems define an *event set* structure, which lists hardware events to be counted together. PAPI and MPX implement event sets differently, so these data structures cannot be used interchangeably between them, but the basic ideas are the same for both systems. A program defines an event set by calling a constructor function with a list of events. Programs initiate measurements by passing an event set to a start function, and counting can proceed on multiple event sets at the same time. In PAPI, though, counter conflicts limit which sets can run together. MPX removes this restriction.

In both systems, a program can read or reset counters for any running event set. Programs can specify options for event sets that define certain measurement characteristics, such as whether events that occur in kernel mode are counted. These features are only partly implemented in both PAPI and MPX. Both are thread-safe and can count events separately for different threads. At present, they cannot define process-wide event sets that count all events in all threads. Although MPX mimics the major functionality of PAPI, it does not include corresponding functions for every function in the PAPI library.

4 Multiplexing

Counter multiplexing involves sharing a single counter among several event measurements over a time period T. (T and other times may be measured either in seconds or in processor cycles.) During this time, the counter will be programmed to measure different event types in sequence. Each event type, e, will be measured during a series of time slices. The duration of the i^{th} slice for event e is $s_{i,e}$, and $s_{i,e}$ can vary with i. At the end of each slice, the number of events counted is added to a cumulative total for e, and the counter will begin counting a new event during the next time slice. For each event type, the user would like to know N_e, the number of times e occurred during T. If a counter is multiplexed, an exact value for N_e is not available, since some occurrences of e will happen when the counter is recording other event types. However, if e occurs at a reasonably constant rate during T, then N_e can be

estimated as follows: Let C_e be the actual count of e accumulated over T and let $S_e = \sum_i s_{i,e}$; that is, S_e is the total time events of type e were counted. Then

$$N_e \approx C_e \frac{T}{S_e}.$$

Counting C_e is easy, but computing S_e and T can be more difficult, as the following discussion will show.

MPX triggers the switching of counters from one event type to the next using the Unix interval timer and signal handling features. At the beginning of a measurement, MPX calls `setitimer` to deliver a signal after a specified interval (10 milliseconds by default). The `ITIMER_VIRTUAL` flag is set, causing the timer to count time only when the process that initiated it is running.

One event type, e_1, is chosen from the user's event set, and a hardware counter is programmed to monitor e_1. The application then proceeds with its computation. When the timer expires, Unix invokes the signal handler. The handler halts counting of e_1, stores the current count, and starts counting e_2. The timer automatically resets itself. When the last event in the set has been counted, e_1 is scheduled again, and this sequence continues until the measurement period ends.

An early implementation of MPX computed S_e and T by counting the number of times, k_e, that each event e was scheduled to be counted. For each e, it computed $S_e = k_e s$, where s is equal to the fixed timer interval. It then computed $T = \sum_e S_e$. In the final computation of N_e, s appears in both the numerator and the denominator, so its actual value cancels out. The software also made a small correction to account for the partial time slice at the end of a measurement period.

This approach produced acceptable results for simple measurements, but it does not work for measurements in threaded programs. The problem is that many systems have one virtual timer per process, not one per thread. Moreover, the signal generated when the timer expires can be delivered to any thread. These systems have no way to set up separate timers for separate threads. As the next section will show, one signal handler can serve all the threads in a process, but the value of s will vary. Although the timer expires at fixed intervals from the perspective of the process as a whole, multiple threads running on one CPU will have their handlers invoked at unpredictable intervals. The problem is less severe when each thread runs on a different CPU, but variations in the time slice for each thread can still arise from the technique MPX uses to schedule the counters in multiple threads.

To compute S_e accurately, the software must measure $s_{i,e}$ separately for each time slice of each event measurement. MPX does this by counting processor cycles simultaneously with each event. This solution assumes that processors have two or more event counters, and that every counter can measure cycles. An informal survey of current CPUs suggests that these assumptions are valid. Therefore, no matter what event is chosen to run during a given time slice, MPX can always count cycles simultaneously in another counter. Each time the signal handler initiates a different event type measurement, it updates both C_e and a running count of S_e. It also maintains a running count of T. With this information, the software can easily compute an estimate of N_e either after a measurement is complete or while it is still running.

MPX counts only one event type (other than cycles) at a time. An alternative approach would be to schedule multiple nonconflicting event measurements on the available counters during each time slice. Since some event types could then be measured more often, S_e would increase for those event types and possibly produce more accurate estimates of N_e. However, an algorithm for selecting compatible combinations of active event types would need to balance the goal of maximizing counter usage against the need to schedule each event type often enough to produce useful data. Such an algorithm would either need to run at every time slice, or else a fixed schedule of counter usage would have to be computed every time an event set was started or stopped. The potential increase in accuracy to be gained from this design does not appear to justify the added complexity, since, as Section 6 notes, the accuracy is already quite good in most cases.

5 Overlapping measurements

Some measurement tasks require counter measurement periods to overlap. Figure 1 shows three types of overlapping measurements: partial overlap, nesting, and multithreading. Partial overlap occurs when an application initiates two or more separate measurements, and one measurement period begins before a preceding period has ended. Nesting occurs when one measurement period occurs completely within another measurement period. Multithreaded overlap occurs when two or more threads carry out measurements (on the same or different regions of code) concurrently. Individual threads may be scheduled to run on the same or different CPUs, and the software must work correctly and accurately in either case. However, the results of some measurements may well be different in the two cases because threads running in parallel may interact with each other differently from threads scheduled sequentially on the same CPU.

All three cases require a level of abstraction, above the basic hardware, that permits multiple concurrent measurements to use the same counter registers. PAPI and some vendor-specific libraries provide this abstraction. In PAPI, each event set may contain one or more events, up to the total number of counters available on the CPU. Several event

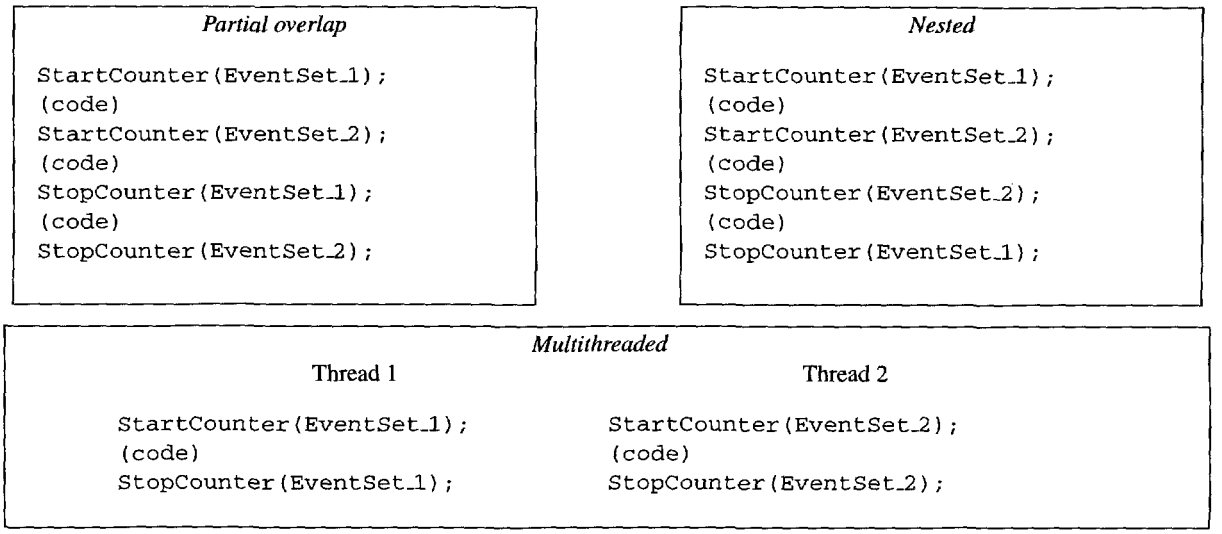

Figure 1. Three types of overlapped counter measurements.

sets can be defined and run in overlapping regions of code, as long as they contain no conflicting events.

Overlapping measurements present two further challenges when the counter library supports multiplexing:

- Each thread must correctly execute the scheduling algorithm that shares the counter registers among the event types being measured.
- The multiplexing library must attribute cycles and events measured during a time slice to multiple event sets.

5.1 Multithreading

MPX has been implemented on an IBM RS6000/SP with four CPUs per node. The system runs AIX 4.3, which has per-process (rather than per-thread) timers and signal handling. Therefore, one timer must serve all the threads in the process.

MPX maintains a list of the threads for which the application has requested counter measurements (called "counting threads"). When the timer expires and triggers a signal, the handler may run in any one of the process's threads. This "timer handling thread" might not be a counting thread, and it might not even be running user code. The handler function will traverse the list of counting threads and send a SIGVTALRM signal to each one (except itself) using the `pthread_kill` function. (Despite its name, `pthread_kill` can send any kind of signal, not just SIGKILL.) The handler will also increment a global counter of threads that should respond to these signals (Figure 2). The timer handling thread will then execute the rest of the handler code, as described below. Each counting thread receiving a signal will activate the same signal handler that the timer handling thread used. However, unlike the timer handling thread, these "receiving threads" must not reissue the signal, or they would create an endless chain of signals. Instead, they examine the global counter; if it is nonzero, a thread will know that it is a receiving thread rather than a timer handling thread. It will decrement the global counter and execute the rest of the handler code. The counter will reach zero when the last receiving thread decrements the global counter. When the timer expires again, the thread responding to the signal will recognize that it is the timer handling thread.

The timer automatically resets itself to deliver another signal after the specified interval has expired again. Potentially, this could happen before all the threads had responded to the previous signal. If the thread arbitrarily chosen by the operating system to handle the signal already has a pending signal from a `pthread_kill` call, the new signal will simply be dropped. If the chosen thread does not have a pending signal, it will invoke its handler and behave as a receiving thread. The global counter will be decremented an extra time in that case, and the last receiving thread to handle its signal will find the global counter has reached zero. It will therefore behave as a timer handling thread and reinitiate the distribution of signals. If timer signals continue to arrive faster than the handlers can operate, the handlers will run correctly, but the program won't get much work done. This situation is unlikely to arise in practice because, as Section 6 shows, the timer interval is typically much larger than the running time of the handler.

In an alternative implementation, the handlers were pro-

```
static int threads_responding = 0
lock( counting_thread_list_lock )
if( threads_responding == 0 ) {      // timer-handling thread
    for each counting thread t other than this thread {
        ++threads_responding
        pthread_kill( t, SIGVTALRM ) // signal thread t
    }
} else --threads_responding          // receiving thread

unlock( counting_thread_list_lock )

if( this thread is counting ) {
    // Code for recording event and cycle counts
    // and switching the current event goes here.
}
```

Figure 2. Pseudocode for managing timer signals in a multithreaded program.

grammed to restart the timer only after the last receiving thread had finished executing its handler. However, in some cases signals were dropped, causing the handlers to cease operating entirely.

5.2 Managing overlap

MPX uses the PAPI infrastructure, which can already handle overlapping measurements. However, handling overlap for multiplexed counters requires additional effort.

In PAPI, if a program starts an event set when another set is already running, PAPI notes which individual events appear in both sets and records their current counts. Each event in the new set that is not already running is assigned to a counter, provided there are no conflicts. If a conflict is found, PAPI returns an error code instead of starting the new set.

MPX uses a "master event list" to manage overlap. This list includes all the events that appear in the MPX event sets created by a particular thread. Each distinct event appears only once in the list, and MPX permits an event set to be used only by the thread that created it. The list maintains a cumulative count of C_e and S_e (the observed count and event measurement time) for each event. Each event also carries a reference count, which indicates how many different MPX event sets use that event, and an "activation" count that indicates how many of these event sets are currently running. An activation count greater than zero makes an event eligible to be scheduled. Each thread has no more than one event designated as the "current event;" this is the event type that the hardware is actually counting at a given moment (along with cycles). If no event set for a thread is active, then there is no current event.

When an MPX event set is started, the activation count for each of its events is incremented, and if there is no current event, one is chosen from this set. MPX also records the current values of C_e and S_e for each event in the set. For events that are not already active, these values will be zero.

When the handler for a thread executes, it first determines whether to send any additional signals, as described in Section 5.1. Then it stops the current event (if any) for that thread, adds the counter value and the cycle count to C_e and S_e, and selects the next active event from the master list. The hardware is programmed to begin counting this new current event. MPX relies on the underlying counter software to gather thread-specific counter data.

When an MPX event set is stopped or read, the software reads the current stored values of C_e and S_e for each event and subtracts the initial values recorded earlier to compute C_e and S_e for the period during which the event set was running. The software also determines whether the thread's current event is among the events in the set being read. If so, the counter and cycle count for that event are read and added to the running total. (A similar correction for the current event is applied when an event set is started.) Each event set also measures T (the total duration of the measurement period) while it is running, and with this information MPX can estimate N_e, the total event count, for each event type as described earlier.

When an MPX event set is destroyed, the reference counts for the corresponding events in the master list are decremented, and any event type whose reference count reaches zero is removed from the master list.

6 Performance

This section describes the accuracy and the overhead of the MPX measurements.

The default time slice t for multiplexing is 10 milliseconds. (In the version of AIX used for these tests, nonprivileged users cannot set shorter timer intervals.) However, there is no guarantee that the actual interval for each counter measurement will be close to this value because of the scheduling complexities described in Section 4. The length of a time slice limits the granularity of measurement for MPX. If the measurement period T is less than the time slice, only the first event type scheduled will be measured; other event types will not be counted. In general, for event sets that contain n event types, measurement periods of $T \leq t(n-1)$ will not produce data for some events.

Figure 3 shows the accuracy of the multiplexed measurements compared to the nonmultiplexed measurements for four event types. These measurements were taken as follows: a series of loops performing floating-point multiplication and addition were run for specified numbers of iterations, ranging from 2^{17} to 2^{24} by whole powers of two. For a 10 millisecond time slice on the 332 MHz test system, $t = 10\text{ ms} \times 332\text{ MHz} = 3.32 \times 10^6$ cycles. For four multiplexed events, $n = 4$, so a measurement period of $T \geq tn = 13.3$ million cycles will produce complete data. The cycle counts for the 2^{17}-iteration loops ranged from 5.3 million to 8.6 million, so tests on loops of this size produced valid data for only some of the events; the rest were reported as zero. The computations used data stored in large arrays, and three separate loops types were executed: one was tiled to maximize L1 cache utilization, one was untiled, and one used random indirect array references to minimize cache reuse. For each loop, the test program counted total cycles, L1 data cache misses, total floating point operations (FLOPs), and number of load requests. First, MPX measured all these events concurrently, and then the tests were repeated with PAPI measuring each event type separately. For the indirect addressing measurements, the test software used the same sequence of indirect array references for both the MPX and the PAPI measurements. The graphs show the ratio of the values measured using MPX to the corresponding values measured using PAPI alone. Although there are a few outliers, especially for lower numbers of events and for multithreaded measurements, the MPX measurements were usually within 5% of the PAPI numbers. Frequently, the results agreed within 1%.

The most significant inaccuracies appeared in the L1 cache miss measurements for the tiled loops. MPX overestimated the counts by up to 70%. These errors are probably due to cache pollution by the MPX handler software. Each time the handler runs, it evicts some data from the L1 cache. When the handler returns, the computation loop incurs numerous cache misses that would not have occurred if the handler had not run. Of course, these cache misses also occur for the untiled and indirect addressing loops, but the latter two loops would have incurred cache misses anyway because they are not tuned for good cache utilization. Therefore, the number of *excess* misses due to cache pollution by the handler is much greater for the tiled loops. The exact number of excess cache misses varies widely, but typical values are in the range of a few thousand per execution of the handler for the tiled loops and few hundred for the other loops. The number of excess cache misses grows for the tiled loops with longer runs because each execution of the handler causes a new set of cache misses, whereas without the intrusion of the handler, the tiled loop would normally incurs many misses initially as it brought data into the cache, but then it would reach a steady state with far fewer misses. Inaccurate cache miss measurements for computations with highly tuned cache usage is the main drawback of the MPX multiplexing software.

For the multithreaded tests, multiplexing was somewhat less accurate than for single threading, but still generally quite good (Figure 4). The cycle count data appears to show significant inaccuracy for all measurement types, but this data is misleading. For the multithreaded tests, the actual run times of the tests do vary somewhat. Since the MPX and PAPI cycle counts are for different runs, the disagreement between the two is due to actual differences in run time, not inaccurate measurement. In fact, the cycle count data for MPX is very accurate because MPX counts cycles continually in one of the registers. Any differences between MPX and PAPI should be due only to the overhead of the MPX software (discussed at the end of this section). Separate tests were conducted in which the cycle count was measured using PAPI and MPX simultaneously during an MPX multiplexed measurement. (This was possible because the PowerPC 604e can monitor cycles in any of its four counters; MPX used two of these registers at a time, and PAPI used one of the others.) In these tests, the PAPI and MPX cycle counts agreed within 1%.

The main overhead of the MPX software is in the handler that responds when the timer expires. For single-threaded programs, this handler executed in about 160 microseconds on the test system, and varying the number of events multiplexed didn't change this time significantly. (With no events being counted, the overhead dropped to 50 microseconds or less.) For programs with four threads on a four-CPU node, the handler overhead varied from about 170 to 215 microseconds. These numbers do not include the overhead of the system software that invoked the signal handler. Not all of the handler time appears in the cycle count for a measured program, because the hardware counters are stopped partway through the handler and restarted near the end. The intervening computation to update various counts and select the next current event happen "off the clock." Of course, the total overhead does add to the wall-clock time of the program, but since the handler runs only about once every 10 milliseconds, the effect on run time is small.

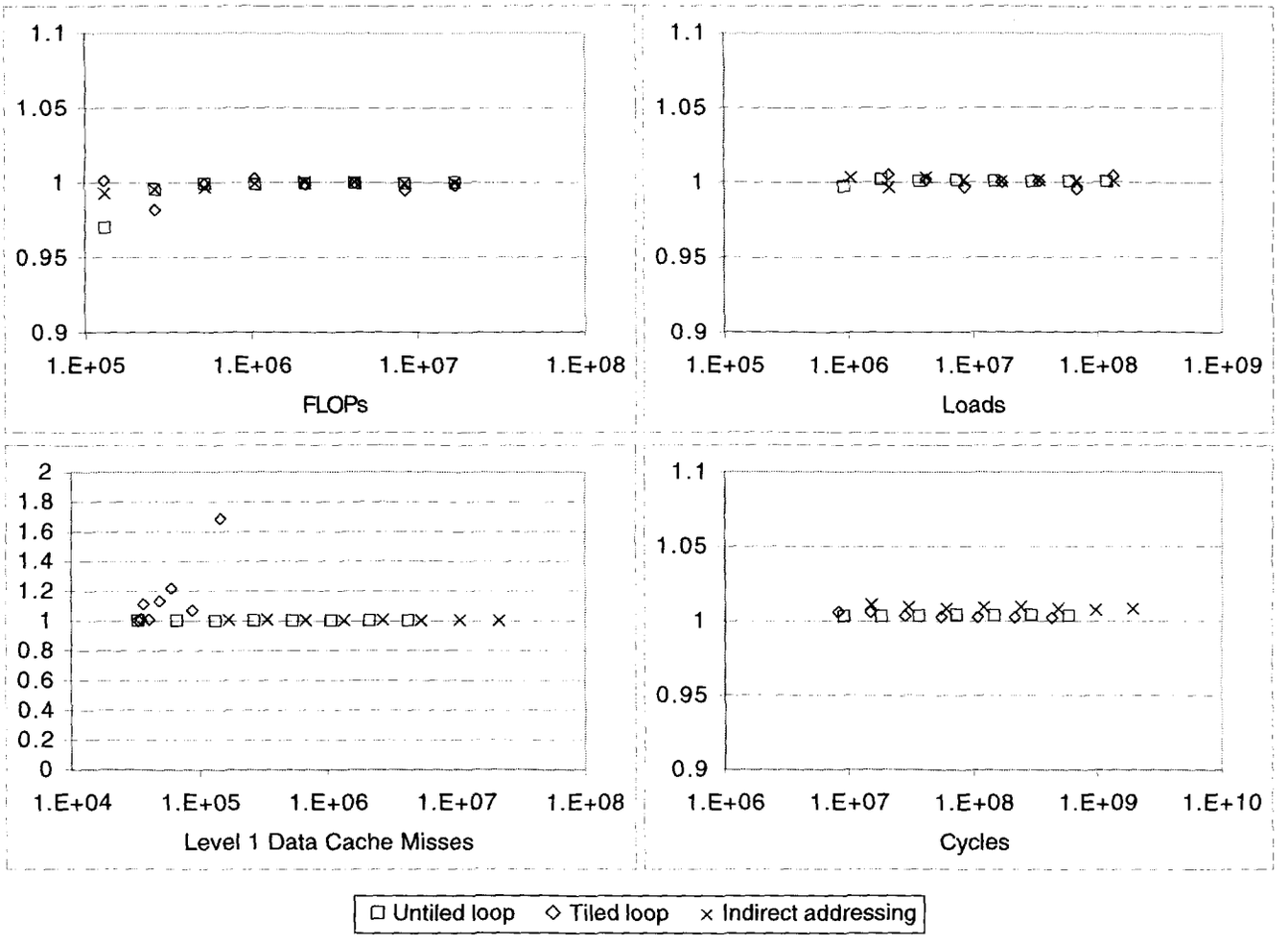

Figure 3. Accuracy of MPX measurements compared to PAPI-only measurements as a function of the number of events counted. Values greater than one indicate that MPX overestimated the event count; values less than one indicate underestimation. Measurements in this figure were made on a single-threaded program.

7 Conclusions

Ideal hardware performance counters would be able to measure any combination of events concurrently. However, since most current CPU designs lack this flexibility, multiplexing is a useful alternative. For code regions that run for a reasonably long time, multiplexing can produce accurate performance data with minimal overhead.

The software described in this paper implements multiplexing on top of the PAPI library, and it supports partly-overlapped, nested, and multithreaded measurements. Future enhancements to this software may include implementing more PAPI functionality (or merging the software into PAPI) and improving accuracy for short measurement periods and multithreaded programs.

Acknowledgments

I thank Phil Mucci and the PAPI team for developing the software on which MPX is based. Phil was very helpful in discussing approaches to multiplexing, answering questions, and responding to problem reports. Deborah Walker and Bronis de Supinski of Lawrence Livermore National Laboratory made many useful comments on early drafts of this paper. Bronis also suggested improvements in the data collection techniques.

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. UCRL-JC-140186.

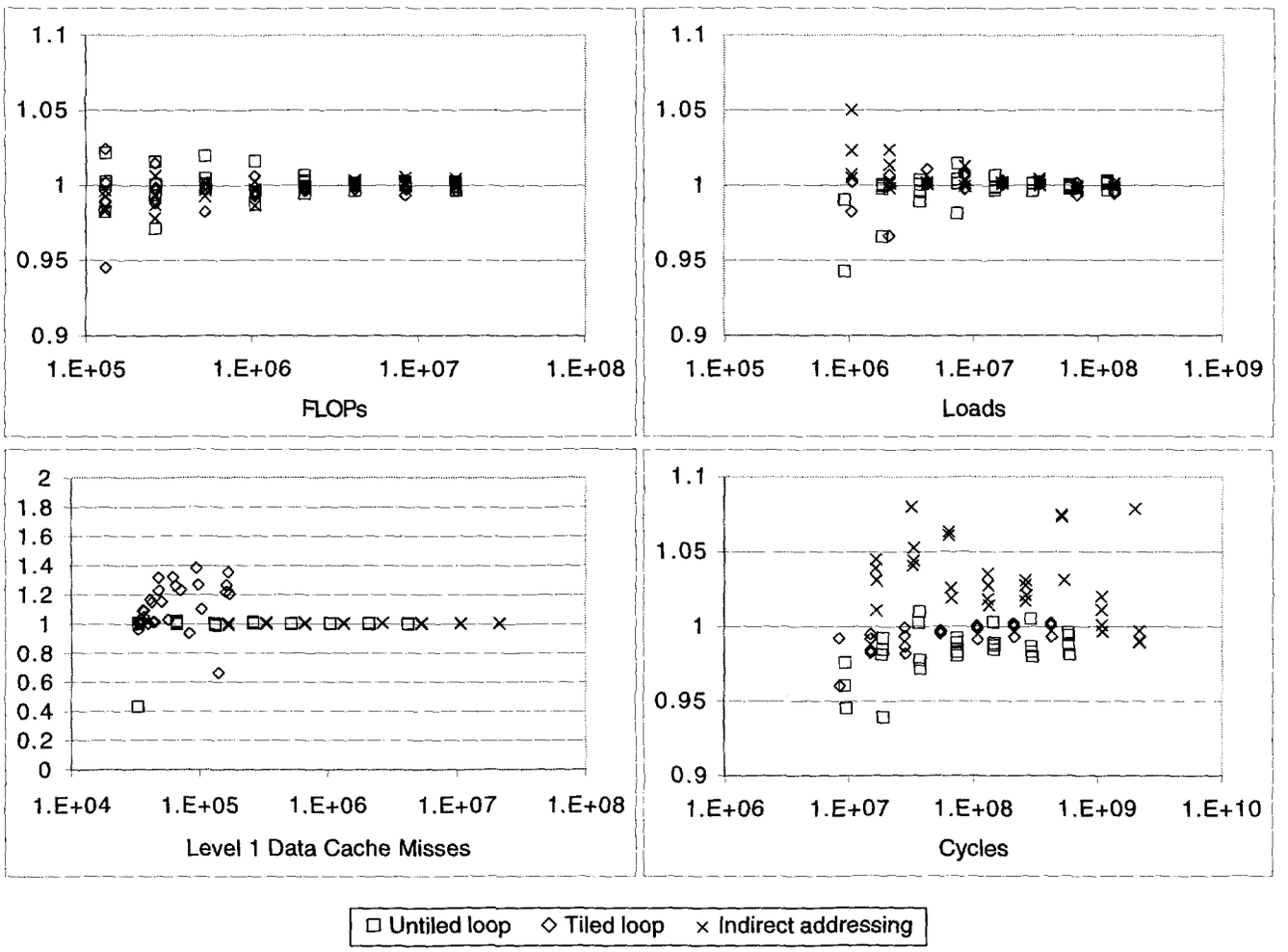

Figure 4. Accuracy of MPX measurements for a program running four threads. See the text for a discussion of the cycle count data.

References

[1] R. Berrendorf and H. Ziegler. *PCL—The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 1.3).* Central Institute for Applied Mathematics, Research Centre Jülich GmbH, Jülich, Germany, November 1999.

[2] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, pages 292–302, December 1997.

[3] F. E. Levine and C. P. Roth. A programmer's view of performance monitoring in the PowerPC microprocessor. *IBM Journal of Research and Development*, 41(3), May 1997.

[4] Motorola, Inc. *PowerPC 604e RISC Microprocessor User's Manual*, 1998.

[5] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Department of Defense HPCMP Users Group Conference 1999*, June 1999. `http://icl.cs.utk.edu/projects/papi/dodugm99/papi.html`.

[6] Performance data standard and API. `http://icl.cs.utk.edu/projects/papi/`, June 2000.

[7] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of Supercomputing '96*, November 1996.